Exact Synthesis For Logic Synthesis Applications With Complex Constraints

Eleonora Testa*, Mathias Soeken*, Odysseas Zografos†‡, Francky Catthoor†‡, and Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Switzerland
†KU Leuven, Belgium
‡IMEC, Belgium

***Abstract*—Exact synthesis is the problem of finding logic networks that represent given Boolean functions and respect given constraints. With exact synthesis it is possible to find optimum networks, e.g., in size or depth; consequently, it primarily finds application in logic optimization. However, exact synthesis is also very helpful in logic synthesis applications necessitating complex constraints that are present in the hardware primitives or the logic representations for which the synthesis has to be performed. Conventional heuristic logic synthesis algorithms are not considering such constraints. They still can be employed to optimize networks, but they cannot guarantee that optimized networks meets all requirements.**

Being faced with a logic synthesis application that seeks for low-depth majority-based networks with limited fan-out for small functions, we demonstrate how state-of-the-art exact synthesis algorithms can be adapted and used to find logic networks that match these constraints. To emphasize the need for exact synthesis, we also demonstrate how conventional logic synthesis either fails to find constraint-satisfying logic networks or yields networks of inferior quality.

I. Introduction

The aim of exact synthesis is to find logic networks that represent given Boolean functions under a set of constraints. Exact synthesis is of great relevance when considering logic optimization, since it is able to find optimum networks. Depending on the design and application, optimality is sought with respect to different objectives, e.g., size or depth. Exact synthesis is a special case of the *Minimum Circuit Size Problem* [1], which asks whether a Boolean function g can be realized by a network of size at most r; it is considered an intractable problem [2]. Due to its complexity, exact synthesis is tipically used to solve problems of limited size, i.e., functions with about 8 variables.

Exact synthesis plays a key role in logic synthesis applications that need to take into account complex constraints. Many beyond-CMOS technologies have been studied in the last decade as replacement or enhancement for CMOS. Some examples are *Quantum-dot Cellular Automata* (QCA, [3]) or spin-based devices, such as *Spin Wave Devices* (SWD, [4]) and *Spin Torque Majority Gate* (STMG, [5]). A broad variety of technologies has resulted in many and diversified constraints which need to be taken into account by novel logic synthesis tools. As an example, several emerging nanotechnologies do not have an efficient inversion implementation or have limited fan-out capabilities. Some technologies have more than one constraint that needs to be respected at the same time. These constraints are often present due to restrictions in the hardware primitives or the logic representations for which the synthesis has to be performed. Classical heuristic logic synthesis tools are not taking into account such constraints. They could be used in the optimization process, but they may lead to solutions which do not meet all the requested constraints. Moreover, no solution may exist if constraints are too tight and heuristic optimization algorithms cannot identify this.

In this paper, we illustrate the use of exact synthesis for logic synthesis applications that deal with many and diversified technological constraints. We consider small multi-outputs functions (i) based on majority that necessitate (ii) limited-depth and (iii) restricted fan-out for each node. These requirements are motivated by an application in industrial project in which these small functions are the result of a pre-partitioning process. We demonstrate that state-of-the-art exact synthesis algorithms can be adapted to solve complex constraint-problems. Exact synthesis algorithms can be implemented in different ways [6], [7], [8]. We use a Boolean Satisfiability (SAT) formulation based on [9]. *Majority Inverter Graphs* (MIGs, [10], [11]) are used as underlying logic representation to our exact synthesis. A MIG is a data structure for Boolean function representation and optimization based on 3-input majority $\langle xyz \rangle$ and inversion. To highlight the importance of exact synthesis for these constraint problems, we demonstrate that conventional synthesis tools are not able to find optimum circuits that meet all the constraints or produce circuits with lower quality. Furthermore, exact synthesis is useful in understanding if a solution exists or if the given constraints are too restrictive.

II. Exact Synthesis

In this section, we describe the SAT formulation proposed by Knuth [9] to find the area-optimum normal Boolean network for functions $g_1, \ldots, g_m$ which depend on n variables. This SAT encoding has been inspired by the work of Kojevnikov et al. [7] and Éen [12]. Recently, the formulation has been extended by Soeken et al. [13] for combinational delay optimization.

Given a function g of n inputs $x_1, \ldots, x_n$, a Boolean network is defined as a sequence of 2-inputs gates $(x_{n+1}, \ldots, x_{n+r})$, where for each gate i:

$$x_i = x_{j(i)} \circ_i x_{k(i)} \tag{1}$$

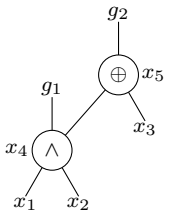

Fig. 1. Example of Boolean network, $x_4 = x_1 \wedge x_2$ and $x_5 = x_3 \oplus x_4$.

with $n < i \leq n + r$. In other words, the two inputs of each gate i are previous gates or inputs. The $\circ_i$ represents one of the 16 binary operations. A Boolean function g is called *normal* if $g(0, \ldots, 0) = 0$. If all the gates of a Boolean network are normal, then the network represents a normal Boolean function. For a normal Boolean network, each 2-input gate can represent 8 out of the 16 possible binary functions.

Knuth's idea is to verify if it is possible to realize functions $g_1, \ldots, g_m$ with a normal Boolean network of size r. In the following, variables and clauses proposed by Knuth are illustrated.

Variables: Let r be the number of gates, m be the number of outputs, and n be the number of inputs. Then, the variables used for the SAT formulation are:

$$\begin{aligned} x_{it}: &\quad t^{\text{th}}\text{bit of } x_i\text{'s truth table} \\ g_{hi}: &\quad [g_h = x_i] \\ s_{ijk}: &\quad [x_i = x_j \circ_i x_k] \text{ for } 1 \leq j < k < i \\ f_{ipq}: &\quad \circ_i (p, q) \text{ for } 0 \leq p, q \leq 1, p + q > 0 \end{aligned} \tag{2}$$

with $1 \leq h \leq m, n < i \leq n + r$, and $0 < t < 2^n$. For each gate x_i, the variable x_{it} represents the value of t^{th} bit in the truth table. Each output variable g_{hi} is true if the function g_h is represented by the gate x_i. The select variable s_{ijk} encodes the children of node x_i. The variable is true if gates x_j and x_k are the children of gate x_i. In this scenario, $\circ_i$ is one of the 8 normal 2-input Boolean functions. The variable f_{ipq} encodes the operation of gate x_i. This is true if for the input assignment (p, q), the operation x_i evaluates to true. It is important to highlight that this method works for normal Boolean functions. If a function is not normal, we find the optimum network for the inverted function. At the end, we invert the output node in order to obtain the original function. The normal property allows Knuth to ignore x_{i0} and f_{i00} for each i.

In the following, we illustrate an example taken from [13] to explain this SAT formulation and, in particular, the variables assignment. We consider the network shown in Fig. 1, with inputs x_1, x_2 and x_3, therefore $n = 3$. In this example, $r = 2$, $x_4 = x_1 \wedge x_2$ and $x_5 = x_3 \oplus x_4$. The gate index i ranges from 4 to 5. Variable x_{it} encodes the truth table for each function of the multi-outputs network. Since $n = 3$ and since we know that $g(0, \ldots, 0) = 0$ (g is normal), the truth table bit t ranges from 1 to $2^n - 1 = 7$.

t	=	7	6	5	4	3	2	1
x_{4t}	=	1	0	0	0	1	0	0
x_{5t}	=	0	1	1	1	1	0	0

Since we are considering a multi-output network, each gate could be an output. Two out of the four output variables are assigned to 1, since $g_1 = x_4$ and $g_2 = x_5$.

$$g_{14} = 1,\ g_{15} = 0,\ g_{24} = 0,\ g_{25} = 1$$

There are three select variables for $i = 4$ and six for $i = 5$. For each gate, only one select variable is equal to 1. For instance, variable $s_{412} = 1$, since x_1 and x_2 are children of node x_4.

k	=	2	3	4
s_{41k}	=	1	0	
s_{42k}	=		0	
s_{51k}	=	0	0	0
s_{52k}	=		0	0
s_{53k}	=			1

Finally, the AND and XOR operations are encoded in the f_{ipq} variables. For this example:

p, q	=	0,1	1,0	1,1
f_{4pq}	=	0	0	1
f_{5pq}	=	1	1	0

Clauses: In order to have a working algorithm, some clauses need to be added:

- a main clause that describes how truth tables are computed for each gate, depending on children (s_{ijk}) and operation (f_{ipq}):

$$\begin{aligned} \big(s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})\big) \rightarrow (f_{ibc} \oplus \bar{a}) = \\ (\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a})) \end{aligned} \tag{3}$$

- a clause to constrain each output value to be the same as the one of the gate it points to;
- a clause to state that each output is realized by one gate in the network;
- and a clause to have two inputs for each gate.

In addition to the mandatory clauses listed above, some auxiliary clauses can be added to reduce the solving time of the SAT solver. More details about both clauses formalization and additional clauses can be found in [9], [13].

III. Constraints Encoding

In this section, we illustrate how state-of-the-art exact synthesis algorithms can be adjusted to solve constraint-problems. Knuth's algorithm is used to find the optimum normal Boolean network for functions $g_1, \ldots, g_m$. In our case, we make use of MIGs as data structure for exact synthesis. Some changes to the original algorithm are then necessary in order to extend our analysis to 3-input majority gates. Further, some additional constraints need to be considered both for the maximum depth and for the maximum fan-out. We demonstrate that Knuth's algorithm can be adapted to work with 3-input majority gates, and to limit depth and fan-out for multi-ouputs networks.

A. 3-input Majority Gates Constraint

Here, the extension to 3-input gates and the restriction to only majority gates is illustrated.

The x_{it} and g_{hi} variables are used in the same way as proposed by Knuth. They encode the truth table and the output gates, respectively. Since we are working with 3-input gates, both the s_{ijk} and f_{ipq} need to be reexamined. Each select variable should consider three different children, here called x_j, x_k, and x_l. The select variables s_{ijkl} is true if the operands of gate x_i are x_j, x_k, and x_l. In a similar way, the function variables should take into account the 3-input operations. The variable f_{ipqu} is true if the operation of gate x_i is true under the input assignment (p, q, u).

In order to restrict the 3-input operations to only normal majority functions, a list of all 3-input majority truth table has been considered. Being p, q, and u the 3 inputs, each gate may realize $\langle pqu \rangle$, $\langle \bar{p}qu \rangle$, $\langle p\bar{q}u \rangle$ or $\langle pq\bar{u} \rangle$. Since the majority operator can behave as AND or OR using constant inputs, also all normal truth tables with constant `1` or `0` have to be considered. In this scenario, also ab, $\bar{a}b$, $a\bar{b}$, and $a + b$ have to be taken into account as possible normal majority operations. The total number of allowed operations is then equal to 8. However, the variables f_{ipqu} allow for a representation of all 128 normal 3-input functions. For each gate i, the operation variable o_{iw} is true if the operation of gate i is w, where w is one of the 8 possible normal majority operations.

Two clauses need to be added. First, a given f_{ipqu} implies a different operation w. For instance, if for gate i it holds that:

p, q, u	=	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
f_{ipqu}	=	0	0	1	0	1	1	1

then the operation $\langle pqu \rangle$ is implemented. Being $\langle pqu \rangle$ the operation with $w = 1$, then the following constraint is added:

$$\begin{aligned}&(o_{i1} \rightarrow (\bar{f}_{001} \wedge \bar{f}_{010} \wedge f_{011} \wedge \bar{f}_{100} \wedge f_{101} \wedge f_{110} \wedge f_{111})) \\ &= (\bar{o}_{i1} \vee (\bar{f}_{001} \wedge \bar{f}_{010} \wedge f_{011} \wedge \bar{f}_{100} \wedge f_{101} \wedge f_{110} \wedge f_{111}))\end{aligned} \tag{4}$$

For each gate i, (4) is added for each operation w. Further, clause $\bigvee_{w=1}^{8} o_{iw}$ ensures that each gate realizes at least one of the 8 operations.

Both Knuth's algorithm and the one presented in [13] work with normal Boolean network with 2-inputs gates. Previous work has considered 3-input gate [14]. A dedicated MIG encoding could have been considered, as in [15]. Neverthless, here the aim is to demonstrate that existing algorithms can be adapted to solve complex constraints problem. In our case, we easily adjust existing algorithms, without changing clauses and with minor changes in the variables encoding.

B. Depth Constraint

We need to constrain the maximum depth of the network. The SAT solver should check whether there exists a MIG with r gates that can realize functions $g_1, \ldots, g_m$ with a depth less or equal to Δ. All input arrival times are considered be 0. For each gate i, a variable d_i takes into account the depth of gate x_i with $n < i \leq n + r$. Each variable d_i has a value in the range $0 \leq d_i \leq (i - n)$. The idea is the same as the one proposed in [13], and the depth variable is encoded using the *order encoding* [9]. In this encoding, each value x in $0 \leq x \leq M$ is represented by a bitstring of length M. In particular, it is represented by x ones followed by $(M - x)$ zeros. To make use of order encoding, each depth variable is a bitstring and it is encoded as d_i^ℓ, where $1 \leq \ell \leq (i - n)$.

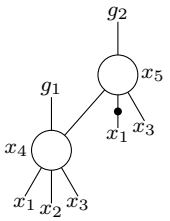

Fig. 2. Example of MIG. Bubbles represent complementation of the edge

The minimum delay of gate x_i is the maximum delay of its children raised by 1. All inputs have a delay of 0, then for $j, k, l \leq n$ the d_i^ℓ variable has value equal to 0. The added clauses are:

$$\begin{aligned}&\bigwedge_{\ell=1}^{j-n} (\bar{s}_{ijkl} \vee \bar{d}_j^\ell \vee d_i^{\ell+1}) \wedge \\ &\bigwedge_{\ell=1}^{k-n} (\bar{s}_{ijkl} \vee \bar{d}_k^\ell \vee d_i^{\ell+1}) \wedge \\ &\bigwedge_{\ell=1}^{l-n} (\bar{s}_{ijkl} \vee \bar{d}_l^\ell \vee d_i^{\ell+1})\end{aligned} \tag{5}$$

The clause $\bar{g}_{hi} \vee \bar{d}_i^\Delta$ ensures a depth $\leq \Delta$, by assigning 0 to the Δ^{th} bit in the order bitstring.

C. Fan-out Constraint

To constrain the maximum fan-out of each node, we make use of cardinality constraints. The cardinality constraint over a set of Boolean variables is a constraint on the number of variables that can have values equal to 1. In particular, we implement the cardinality constraint as proposed in [16].

In our case, the constraint is on the fan-out of each node to be at maximum Φ. Select variable s_{ijkl} encodes that gates x_j, x_k, and x_l are the children of node x_i. To consider the fan-out of node i, we need then to take into account nodes with index larger than i: $s_{i+1jkl}, \ldots, s_{i+njkl}$. Among all these select variables we need to force a constraint on the ones that use i as children. In other words, all the select variables of index larger than i, in which j or k or l is equal to i. Also the output variables g_{hi} need to be considered for this fan-out count.

Fig. 2 shows an example of MIG exact synthesis. x_1, x_2 and x_3 are the inputs of the network, and $n = 3$. As in the previous example, $r = 2$; $x_4 = \langle x_1 x_2 x_3 \rangle$ and $x_5 = \langle x_4 \bar{x}_1 x_3 \rangle$. The gate index i ranges from 4 to 5. Variable x_{it} encodes the truth table for each output of the multi-outputs function. Further,

$$g_{14} = 1,\ g_{15} = 0,\ g_{24} = 0,\ g_{25} = 1$$

There is only one select variables for $i = 4$, which is s_{4123}. There are three select variables for node 5. For this gate, only one select variable is equal to 1: $s_{5134} = 1$, since x_1, x_3, and x_4 are children of node x_5. Finally, the two different majority operations are encoded in the f_{ipqu} variables. For this example:

p, q, u	=	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
f_{4pqu}	=	0	0	1	0	1	1	1
f_{5pqu}	=	1	0	0	1	1	0	1

Only one operation variable o_{iw} is equal to 1 for each node. For the depth, each variable d_i^ℓ has $1 \leq \ell \leq (i-3)$. It follows that for node 4, the depth variable consists of only 1 bit. d_5^ℓ is made of 2 bits, and it can have depth value of 0, 1, and 2. The fan-out constraint on node 4 consists of a cardinality constraint of type:

$$s_{5124} + s_{5134} + g_{14} + g_{24} \leq \Phi \tag{6}$$

where all nodes that use 4 as child are considered.

IV. Exact Algorithms

This section describes the implemented exact synthesis algorithm. It also illustrates three alternatives to the main algorithm.

The algorithm finds a MIG, if this exists, that satisfies all the constraints discussed in Section III. The names of the variables are the ones used in previous sections. The input of the algorithm is the n-inputs m-outputs function g represented as truth tables obtained from the MIG that needs to be optimized. The algorithm starts by trying to find a solution using $r = m$ (assuming that each output represents a different function). If a solution exists for r gates, the algorithm returns a MIG that meets all the requirements, otherwise it looks for a solution with larger size. The algorithm increases the number of gates until the upper bound is reached. If no solution can be found up to the upper bound, it can be concluded that no network exists that meets all the constraints. Let m be the number of outputs, an upper bound for the number of gate r is $13m$. This result is obtained considering that each output could be represented as a tree, with no sharing edges between them. Each tree has one gate on the first level, 3 gates on the second level and 9 gates on the third one, thus 13 gates at most.

The algorithm is described as function $\mathrm{FindMIG}()$ in Alg. 1. First, the SAT solver is instantiated (line 2 in Alg. 1). Then, the algorithm adds all the variables discussed in Section II and III; they include the variables from Knuth's formulation, but also variables d_i^ℓ and o_{iw}. All clauses are then added (lines 4–12). The main clause is the one which encodes the truth table of the circuit (3); this is added for each bit t of each truth table. Other clauses consist of both necessary and additional clauses proposed in [13]; depth, operations, and fan-out clauses are the ones discussed in Section III. The fan-out clause constraints the fan-out of each node i of the network.

Alternative implementations to Alg. 1 are possible. All algorithms take into account the same constraints, however, they may show different performances. We rewrite Alg. 1 by making use of *Counter-Example-Guided Abstraction Refinement* (CEGAR). The idea is to overapproximate the solution space by discarding several clauses, thereby decreasing solving time of the SAT solver. Alg. 2 illustrates one CEGAR version of Alg. 1. Alg. 2 does not add the main clause (3) which encodes the multi-output function g. In this way, the SAT solver may find a solution which does not coincide with g for all inputs assignment t. If this is the case, a refinement of the solution is pursued (lines 13–15). In order to ensure the same functionality, the main clause (3) is added for the first bit t of the truth table that does not agree with g. The updated problem is solved again by keeping the state of the SAT solver active (incremental SAT). This procedure is repeated until the truth tables coincide. During this refinement process, two possibilities emerge:

1) The SAT solver converges to a solution which respects the functionality;
2) The SAT solver is not able to find a solution which respects the new clauses. In this case, the size r is increased and a new solution is searched.

CEGAR can also be applied to abstract the fan-out clauses. First, a solution without fan-out constraints is found; then, the algorithm checks whether a gate i exists that does not respect the fan-out constraint. If it exists, the fan-out constraint is added only for gate i, which has fan-out $> \Phi$. The algorithm is not reported here, since it is similar to Alg. 2.

Both CEGAR methods can also be applied at the same time. We call this method *DoubleCEGAR* (DCEGAR) and it is shown in Alg. 3. In this approach, both the truth table and fan-out clauses are not added in the main function. If a solution of size r exists, the functionality is checked (line 8). If the functionality is respected, then the algorithm ensures that also the fan-out constraint is met (line 9). If both are respected, the MIG is returned (line 10). Otherwise, first the algorithm tries to meet the truth table constraint (lines 15–18) and then the fan-out one (lines 12–13). The algorithm works in a similar way as Alg. 2; if at some point the SAT solver cannot find a

```
1  Function FindMIG(g, Δ, Φ, r)
2      set S ← SATSolver();
3      AddVariables(S, g, Δ, Φ, r, );
4      foreach 0 < t < 2^n do
5          AddMainClause(S, g, t);
6      end
7      AddOtherClauses(S, r);
8      AddOperationClause(S, g, r);
9      AddDepthClause(S, Δ);
10     foreach n < i ≤ n + r do
11         AddFanOutClause(S, Φ, i);
12     end
13     if Solve(S) then
14         return MIG;
15     else
16         return FindMIG(g, Δ, Φ, r + 1);
17     end
```

Algorithm 1: Function 'FindMIG()'

```
Function FindMIG_CEGAR(g, Δ, Φ, r)
    set S ← SATSolver();
    AddVariables(S, Δ, Φ, r);
    AddOtherClauses(S, r);
    AddOperationClause(S, g, r);
    AddDepthClause(S, Δ, r);
    foreach n < i ≤ n + r do
        AddFanOutClause(S, Φ, i);
    end
    while Solve(S) do
        if Functionality_Respected(g) then
            return MIG;
        else
            set t ← first bit which does not respect
            functionality;
            AddMainClause(S, g, t);
        end
    end
    return FindMIG_CEGAR(g, Δ, Φ, r + 1);
```

Algorithm 2: Function 'FindMIG_CEGAR()'

```
Function FindMIG_DCEGAR(g, Δ, Φ, r)
    set S ← SATSolver();
    AddVariables(S, Δ, Φ, r);
    AddOtherClauses(S, r);
    AddOperationClause(S, g, r);
    AddDepthClause(S, Δ, r);
    while Solve(S) do
        if Functionality_Respected(g) then
            if All_Fanouts() ≤ Φ then
                return MIG;
            else
                set i ← node with fan-out > Φ;
                AddFanOutClause(S, Φ, i);
            end
        else
            set t ← first bit which does not respect
            functionality;
            AddMainClause(S, g, t);
        end
    end
    return FindMIG_DCEGAR(g, Δ, Φ, r + 1);
```

Algorithm 3: Function 'FindMIG_DCEGAR()'

solution due to the new clause, then the algorithm searches for a solution with size $r + 1$.

V. Results

In this section, first, we demonstrate how conventional logic synthesis tools are not suitable for complex constraints-problem; then, we illustrate results obtained with the different algorithms proposed in Section IV. Finally, we discuss the feasibility of our method on larger functions.

We developed a C++ program[1] to implement Alg. 1. The implementation uses one of the SAT solvers implemented in ABC [17]. Motivated by our industrial application, for these experiments we used the maximum depth $\Delta = 3$ and the maximum fan-out $\Phi = 3$. We applied our approach to small arithmetic benchmarks and to some small *hwb* [18] circuits. The 'HWB34' benchmark is a multi-output function containing both hwb3 and hwb4. To emphasize the key role of exact synthesis for complex constraint-problems, we demonstrate that classical logic synthesis tools may fail in finding a solution that meets all the constraints. Results are shown in Table I. We optimized circuits using ABC depth optimization (*'clp; sop; fx; strash; resyn2'*). The results are shown in the first part of Table I; only two circuits out of six meet the depth constraint. The 'FO viol.' column represents the number of nodes that violate the fan-out constraint, thus with fan-out $> \Phi$; for the 'ADDER2x2' benchmark, also the fan-out constraint is not respected. The second block of Table I shows results obtained by analyzing each output separately. Each function has been depth-optimized using exact synthesis proposed in [15]. This approach leads to results that meet our depth constraint, but it is time consuming, since all outputs are analyzed separately. Further, this does not optimize the network considering the common nodes and it does not take into consideration the fan-out constraint. Copies of nodes with fan-out $> \Phi$ have been produced. For this case, the runtime is the sum of the runtimes necessary for each output; the manual work needed to separate and reunite the whole circuit are not taken into account. The third block of Table I shows the results achieved using our exact algorithm approach, disregarding the fan-out constraint. Also in this case, copies of nodes with fan-out $> \Phi$ are introduced. Table I shows that the better results are the ones obtained with the exact method implemented in Alg. 1, therefore considering all the constraints (both depth and fan-out). In this case, there are no nodes with fan-out larger than Φ. As an example, ADDER2x2 leads to a better result when also fan-out constraint is added. For this benchmarks, both the exact solutions (disregarding and considering fan-out) lead to a depth equal to 3 and size equal to 6. But for the first case, a copy of one node needs to be introduced since its fan-out is larger than Φ; producing in this way a size of 7.

We applied the four alternatives of Alg. 1 to the same circuits; results are listed in Table II. The first algorithm is the one without CEGAR approach. The second one is Alg. 2, while the third one is the one in which the CEGAR method is applied not considering the fan-out clause. DOUBLE CEGAR is the last method in Table II. The runtimes of the four methods are similar. This is not surprising, since the number of inputs in the considered benchmarks is small. It is important to highlight that we are not optimizing the depth, but just constrain it to be $\leq \Delta$. For the BITCOUNTER3, exact solutions with different depths are found. An extension that considers multi or all exact solutions can be easily included in the algorithm.

The constraints used so far are motivated by an industrial application in which each small function is part of a larger function; and each function should meet the depth and fan-out requirements. To validate the feasibility of this method, we map networks using LUTs of different size; then we apply the SAT-based method on each LUT. We are interested in finding

[1]The code is available online: github.com/eletesta/cirkit-addon-mign-sat

TABLE I
CLASSICAL HEURISTIC AND EXACT SYNTHESIS COMPARISON

Benchmark	I/O	Classical Heuristic ABC				Exact [15] - separated outputs				Exact - no fan-out				Alg. 1		
		Depth	Size	FO viol.	Time [s]	Depth	Size	FO viol.	Time [s]	Depth	Size	FO viol.	Time [s]	Depth	Size	Time [s]
ADDER 2x2	4/3	**4**	11	**1**	0.14	3	7	**1**	0.42	3	6	**1**	1.15	3	6	0.56
MULT 2x2	4/4	3	8	—	0.14	3	12	—	0.75	3	8	—	76.84	3	8	68.92
BITCOUNT3	3/2	**4**	8	—	0.14	2	3	—	0.09	3	3	—	0.00	2	3	0.05
HWB3	3/1	2	3	—	0.13	2	3	—	0.00	2	3	—	0.00	2	3	0.05
HWB4	4/1	**4**	8	—	0.14	3	5	—	0.25	3	5	—	0.25	3	5	0.23
HWB34	4/2	**4**	11	—	0.14	3	7	—	0.84	3	6	—	2.16	3	6	1.97

TABLE II
COMPARISON OF DIFFERENT EXACT ALGORITHMS

Benchmark	I/O	Alg. 1 - All clauses			Truth Table CEGAR			Fan-out CEGAR			DCEGAR		
		Depth	Size	Time [s]	Depth	Size	Time [s]	Depth	Size	Time [s]	Depth	Size	Time [s]
ADDER 2x2	4/3	3	6	0.56	3	6	0.73	3	6	1.33	3	6	0.93
MULT 2x2	4/4	3	8	68.92	3	8	77.58	3	8	76.01	3	8	94.26
BITCOUNT3	3/2	2	3	0.05	3	3	0.05	3	3	0.05	3	3	0.05
HWB3	3/1	2	3	0.05	2	3	0.05	2	3	0.05	2	3	0.05
HWB4	4/1	3	5	0.23	3	5	0.30	3	5	0.24	3	5	0.41
HWB34	4/2	3	6	1.97	3	6	4.93	3	6	2.22	3	6	2.24

TABLE III
SAT-BASED ALGORITHM ON LUTS FROM EPFL BENCHMARKS

Benchmark	3-LUTs				4-LUTs				5-LUTs				6-LUTs			
	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO	#LUTs	# SAT	%	# TO
adder	41	41	100	0	185	185	100.0	0	343	342	99.7	1	399	379	95.0	20
arbiter	64	64	100	0	285	285	100.0	0	395	395	100.0	0	419	419	100.0	0
bar	8	8	100	0	8	8	100.0	0	14	14	100.0	0	13	8	61.5	5
cavlc	65	65	100	0	216	216	100.0	0	190	185	97.4	5	139	115	82.7	24
ctrl	24	24	100	0	34	34	100.0	0	26	23	88.5	3	24	19	79.2	5
dec	8	8	100	0	24	24	100.0	0	32	32	100.0	0	40	40	100.0	0
i2c	76	76	100	0	159	159	100.0	0	151	149	98.7	2	145	124	85.5	21
int2float	45	45	100	0	68	68	100.0	0	55	54	98.2	1	40	30	75.0	10
log2	166	166	100	0	986	984	99.8	2	1620	1485	91.7	135	1932	1428	73.9	504
max	47	47	100	0	115	115	100.0	0	142	142	100.0	0	155	154	99.4	1
mem_ctrl	132	132	100	0	723	721	99.7	2	927	921	99.4	6	948	857	90.4	91
mult	132	132	100	0	784	782	99.7	2	1190	1115	93.7	75	1303	1099	84.3	204
priority	30	30	100	0	53	53	100.0	0	61	61	100.0	0	61	61	100.0	0
router	21	21	100	0	25	25	100.0	0	27	27	100.0	0	25	25	100.0	0
sin	142	142	100	0	700	699	99.9	1	958	910	95.0	48	915	751	82.1	164
sqrt	192	192	100	0	1754	1752	99.9	2	4574	4394	96.1	180	5505	4654	84.5	851
square	112	112	100	0	493	491	99.6	2	615	565	91.9	50	692	571	82.5	121
voter	79	79	100	0	194	192	99.0	2	206	192	93.2	14	215	177	82.3	38
Average			100.0%				99.9%				96.8%				86.6%	

#LUTs is the total number of unique LUTs; # SAT is the number of LUTs which are satisfiable; % is the precentage of SAT over the total number of LUTs; #TO is the number of LUTs that are not finished before the timeout.

how many LUTs can be realized as MIGs that meet all the given constraints for depth and fan-out.

We applied this method on circuits from the EPFL benchmarks,[2], using LUT size $k = 3, 4, 5, 6$, respectively. The LUTs and the functions they represent are obtained from *CirKit*[3] using the command *'xmglut'*. The SAT-based method is applied on all the unique LUT functions, using a maximum depth $\Delta = 3$ and a maximum fan-out $\Phi = 3$. The experiments were performed on a computer with Intel Xeon Processor E5-2680 v3, @ 2.5 GHz, 64 Gb RAM, using a timeout of 1 minute for each LUT function. Table III shows the results; in particular it lists the total number of unique LUTs, the satisfiable LUTs and the total timeouts for each LUT size k. Results show that when 3-LUTs are used to map the circuit, all the functions can be realized with circuits of $\Delta \leq 3$ and $\Phi \leq 3$, and with a runtime ≤ 1 minute. For LUTs of larger size, some timeouts are present and not all the LUTs can be finished in less than 1 minute. No conclusions can be drawn on the satisfiability of functions that

[2]lsi.epfl.ch/benchmarks
[3]github.com/msoeken/cirkit

were not concluded in 1 minute: they could be both satisfiable or unsatisfiable. In the worst case scenario (i.e. all timeouts are UNSAT), results for 6-LUTs show that on average 86.6% of LUTs can be realized with the given constraints. In general, a high number of LUTs from EPFL benchmarks can be realized with circuits that meet all the constraints; we are then confident that our method could produce good results when considering partitioned functions.

VI. Conclusion and Future Work

We demonstrated that existing exact synthesis algorithms can be easily adjusted to solve complex constraints problems. We implemented an algorithm that takes into account all the different constraints and we demonstrated that classical heuristic logic synthesis tools may lead to a solution which does not meet all the requirements. Further, we implemented different versions of the same algorithm.

In this work, we mainly took into consideration small circuits. It is important to highlight that in our target technology, the idea is to partition the functionality into smaller blocks. Each small block should match our constraints on depth and fan-out. Future work will investigate more the results on larger functions and it will consist of a mapping algorithm able to build circuits based on smaller blocks that have limited depth and limited fan-out for each node.

References

[1] V. Kabanets and J. Cai. Circuit minimization problem. *STOC*, pages 73 –79, 2000.

[2] C. D. Murray and R. R. Williams. On the (non) np-hardness of computing circuit complexity. *CCC*, pages 365–380, 2015.

[3] C. S. Lent, D. P. Tougaw, W. Porod, and G. H. Bernstein. Quantum cellular automata. *TNANO*, 4(1):49–57, 1993.

[4] A. Khitun and K. L. Wang. Nano scale computational architectures with spin wave bus. *Superlattices and Microstructures*, 38(3):184–200, 2005.

[5] D. E. Nikonov, G. I. Bourianoff, and T. Ghani. Proposal of a spin torque majority gate logic. *IEEE Electron Device Letters*, 32(8):1128–1130, 2011.

[6] R. Drechsler and W. Gunther. Exact circuit synthesis. *IWLS*, 1998.

[7] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev. Finding efficient circuits using SAT-solvers. *SAT*, pages 32–44, 2009.

[8] M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli. Optimizing majority-inverter graphs with functional hashing. *DATE*, pages 1030–35, 2016.

[9] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley, 2015.

[10] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: a new paradigm for logic optimization. *TCAD*, 35(5):806–819, 2016.

[11] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: a novel data-structure and algorithms for efficient logic optimization. *DAC*, pages 194:1–194:6, 2014.

[12] N. Éen. Practical SAT - a tutorial on applied satisfiability solving. *slides of invited talk at FMCAD*, 2007.

[13] M. Soeken, G. De Micheli, and A. Mishchenko. Busy man's synthesis: Combinational delay optimization with SAT. *DATE*, 2017.

[14] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli. Classifying functions with exact synthesis. *ISMVL*, 2017.

[15] M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli. Exact synthesis of majority-inverter graphs and its applications. *TCAD*, 2017.

[16] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *CP*, pages 108–122, 2003.

[17] R. K. Brayton and A. Mishchenko. ABC: an academic industrial- strength verification tool. *CAV*, pages 24–40, 2010.

[18] Hidden-weighted bit (hwb) functions. *IEEE Trans.*, (C-40):208–210, 1991.